IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 10, Issue 1 (Mar. - Apr. 2013), PP 43-50
www.iosrjournals.org

Design of Predicate Filter for Predicated Branch Instructions

Kanmani[1] And Jayaprakash M[2]
[1,2] *kvg college of engineering,sullia,india*

Abstract: *Implementing innovative hardware feature is one of the techniques to increase processor performance. Branch Prediction is a strategy in computer architecture design, for mitigating the cost, usually associated with conditional branches. In this paper, the feasibility of a novel idea called predicate filter is being presented. The predicate filter acts like a preprocessor on the instruction prefetch queue to eliminate in advance the instructions that need not be executed because of invalid predicate. The simulation of the predicate filter design has been carried out, demonstrating the feasibility of predicate filter.*
Keywords: *predicate filter, branch prediction and instruction pipelining.*

I. Introduction

Speed of instruction execution and efficient memory utilization has been the two major objectives in the development of the microprocessor architectures. Memory efficiency is a software process and with the advent of VLSI, memory efficiency has receded to the background and time efficiency is in the forefront. Speed can be improved by implementing innovative hardware features.Over the past several years, strategies to increase microprocessor performance, scalability and lower cost to higher performance computing have focused on finding more Instruction Level Parallelism (ILP). However two difficult problems limit Instruction Level Parallelism (ILP).

1. Branch Instruction. This introduces control dependencies.
2. Memory Latency is the time it takes to retrieve data from the memory.

Difficulty caused by branch instruction can be removed to a large extent by predication. Predication is a method to handle conditional branches. The main idea of the method is that compiler schedules both possible paths of the branch to be executed on the processor simultaneously.Modern microprocessor architecture has advanced to a very complex level. In past few years processors that support predicated instructions have been designed. Predicates are simply tags that permit a program to execute the instruction conditionally depending on the predicates value, which in turn depends on the outcome of a conditional statement.

While implementing the instruction pipeline, it was noticed that in case of an if-then-else statement both the 'if' clause and the 'else' clause entered into the pipeline. In an if-then-else statement one of the paths is always not executed. The processor speed can be enhanced by allowing only that path into the pipeline that must be executed. This observation has lead to the concept of predicate filter.

II. Background And Related Work

Branches seem to be the most straight forward instruction type for a processor, since there is nothing more to do than modify, conditionally or unconditionally, the value of the Program counter. However, this easy to perform operation has turned out to be one of the most serious obstacles to increase the performance of ILP-processor [1]. Strategies to increase microprocessor performance have focused on finding more ILP.

One of the early attempts to minimize the ill effect of branch instruction was branch prediction. Further improvements in this direction leads to a lot of hardware and software complexities. New approaches to overcome branch penalties were put forward by many architects. Lam M.S and Wilson [2] discuss techniques like control dependence analysis, executing multiple flows of control simultaneously and speculative execution that can be used in relaxing the constraints imposed by control flow on parallelism. DeRosa J.A and Levy H. M [3] make a study of the use of delayed branches, the use of one or two instruction branch design, and the use of condition code. Riesman and Foster [4] proposed Eager Execution, where both paths of a branch are taken, if another branch is encountered before the first is resolved, execution also proceeds down both paths of the second branch. Augustus K Uht talks about Disjoint Eager Execution were resources are allocated to the most likely paths to be executed over the entire branch path space. On the same lines Predication is also one of the techniques used to reduce the impact of unpredictable branches.

The predication enables extra parallelism to be introduced into the pipeline. The basic idea is to avoid conditional branches in the architecture by replacing them with conditional operate.

The disadvantage of predicated instruction is that predication transforms instruction from both the taken and the not taken paths into predicated instructions. Thus predication causes the processor to execute an increased number of instructions.

In this paper the simulated predicate filter demonstrates that, only the non-predicated instructions and the instructions with the true predicate value enter the pipeline stage by filtering out the predicate false instructions in the main instruction queue, thus the hardware resources of the pipelined stages can be efficiently utilized.

III. Proposed Filter Design

The basic design of the proposed predicate filter has 2 stages: Instruction fetch Stage, Predicate - Decode stage.

3.1Design Of Instruction Fetch Stage:

This stage consists of 3 components the program counter, the main instruction queue and a 32-bit buffer. For design simplicity it is assumed that the entire program is preloaded into the main instruction memory. The components in this stage are coupled such that the task of fetching an instruction is completed in one clock cycle.

3.2 Design Of Predicate-Decode Stage:

This is the core part of the predicate filter consisting of 2 de-multiplexers, 1 multiplexer, few predicate buffers, non-predicate buffer, control unit, counter and Predicate file. Non-predicated instructions are sent to the non-predicate buffer. Predicated instructions are sent to the corresponding predicate buffers. Based on the signals sent from the control unit the multiplexer selects instructions either from the predicate buffers or from the non-predicate buffer to the pipeline.

3.3 Working Principle

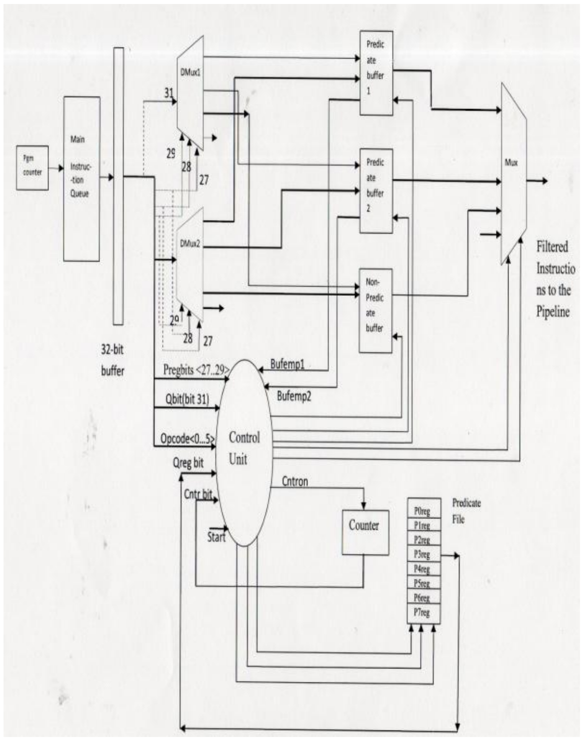


Fig. 1 Predicate Filter Design with Different control Signals

Fig. 1 shows the predicate filter design with different control signals. The program counter in the first stage generates 5-bit sequential address. This 5-bit address is given to the input address lines of 32-bit Main Instruction Queue. The clock signal drives the program counter and for each falling edge of the clock, the program counter generates address.

During the rising edge of the clock, when the read and enable signal of the Main Instruction Queue is high, the instruction from the corresponding address is read out of the instruction queue. For the simulation purpose, read and enable signals of Main Instruction Queue is kept high and the instructions are fetched sequentially. The instruction read out from the main instruction queue is fed as input to the 32-bit buffer.

In the falling edge of the clock, when write and enable signal of the 32-bit buffer is high, the instruction read from the Main Instruction Queue is written into the 32-bit buffer. In the rising edge of the clock pulse, when read and enable of the 32-bit buffer is high the instruction is read from the 32-bit buffer.

The output of the instruction fetch stage is given as input to the predicate-decode stage of the predicate filter. The fetched 32-bit instruction is given to the input of De-Multiplexer2. At same time the 31st bit of the instruction is given to the input of De-Multiplexer1, <27….29> bits of instruction, indicating predicate registers which depends on the compare instruction is given as select pins sel(0), sel(1), sel(2) of the De-Multiplexer1 and De-Multiplexer2 and Opcode bits <0….5> of the instruction is given to the Control Unit.

If the 31st bit of the instruction is '1', that indicates, it is a predicated instruction whose execution depends on the 'compare' result. If the 31st bit of the instruction is '0' that indicates Non-predicated instruction which is to be executed always. In this stage the non-predicated instruction identified by the value '0' in the 31st bit is stored in the non-predicate buffer. If the 31st bit of the instruction is '1' that indicates a predicated instruction. Depending on the value of the bits <27….29> of the instruction, the instruction is sent to the corresponding predicate buffer.

Consider the case where the bits <27…29> has the value "010" and the bit <31> has the value '1' then this predicated instruction is sent to the predicate buffer2.

Role of Dmux1 and Dmux2: The input to the Dmux1 is the bit <31> of the instruction. Output lines of the Dmux1 are given to the 'write' input of predicate buffers. Bits <27…29> of the instruction is connected as select lines of Dmux1 and Dmux2. The input of Dmux2 is the 32-bit instruction. Depending on the bits <27…29>, the instruction is sent to corresponding predicate and non-predicate buffers. Write signal for the non-predicate buffer comes from the control unit.

The Multiplexer at the end is used to read instructions from the different buffers. The select lines of this MUX are controlled by control unit. If only 2 predicate buffers and 1 non-predicate buffer that is a total of 3 buffers are used then only 2 select lines are enough. For more predicate buffers, more number of select lines is needed depending on the number of predicate buffers.

Role of Control Unit: The opcode specified by the bits <0….5> of the instruction is the one of the input to the control unit. The compare instruction has an opcode of "100100". The result of the compare instruction is stored in the predicate file. It is assumed that the result of the compare instruction appears after 6 clock cycles.

For simulation only 2 predicate buffers (predicate buffer1 and predicate buffer2) and 1 non-predicate buffer is considered. In the Multiplexer if the select input is "00" then the instructions from non-predicate buffer is sent as output. If the select input is "01" then the instructions from the predicate buffer1 is sent as output else if the select input is "10" the instructions from the predicate buffer2 is sent as the output of the predicate filter. Predicated instructions with false predicate value are flushed out by resetting the corresponding buffer.

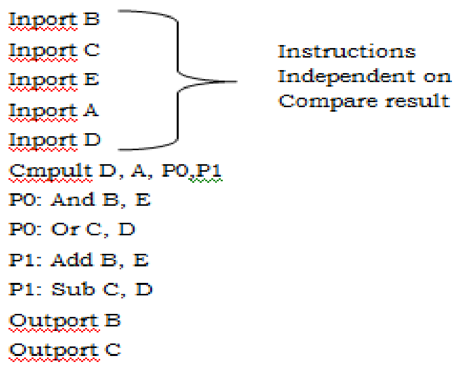
When the compare instruction is fetched, the control unit sends a 'cntron' signal to the counter (the counter acts as a mod-6 counter). After 6 clock cycles the counter sends a signal viz 'cntrbit' to the control unit. In response to this signal the control unit sends enable signal, read signal and the predicate register address as specified by the bits <16….18> of the 'compare' instruction to the predicate register file. The predicate value (Qreg bit) from selected predicate register is read. This value is sent to the control unit. If this bit is '1' then the control signals cntrout(19) and cntrout(18) is made "01" and if this bit is '0' then the cntrout(19) and cntrout(18) is made "10".

The predicate buffers are designed such that when the 'read' address is equal to 'write' address the signal 'bufemp' is set 'high'. This signal is given to the control unit. The moment this signal becomes 'high' the cntrout(19) and controut(18) is made "00". The cntrout(19) and cntrout(18) is fed to the sel(1) and sel(0) of the multiplexerThus, the predicate filter that filters out predicated false instructions from entering the pipeline queue is simulated.

IV. Result Analysis

Consider an example program to store 'and' result in B, 'or' result in C when D < A OR store 'add' result in B, 'sub' result in C when (D > A).

The predicated code of the program is given below.



```
Inport B
Inport C
Inport E
Inport A
Inport D
Cmpult D, A, P0,P1
P0: And B, E
P0: Or C, D
P1: Add B, E
P1: Sub C, D
Outport B
Outport C
```

Instructions Independent on Compare result

Predicated Code

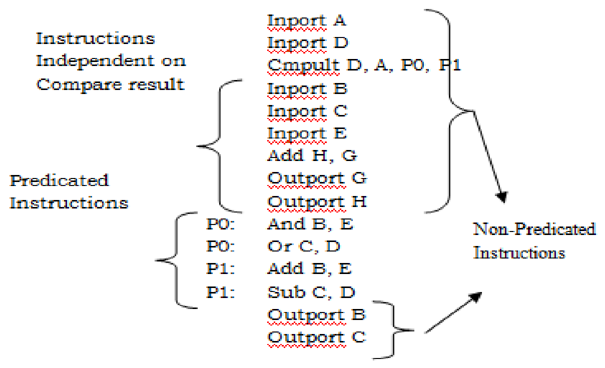
In the program, the ‘compare’ result is known only after 6 clock cycles (Assuming that ‘compare’ takes one clock cycle for fetch, one for decode, one for issue and 3 clock cycles for execution phase. Totally 6 clock cycles are needed for the ‘compare’ result). After 6 clock cycles only the predicated instructions with true predicate value are allowed.

Waiting for compare result in the program execution leads to waste of 6 clock cycles. By placing few independent instructions, those are present before the compare instruction into after the ‘compare’ instruction can avoid waste of clock cycle. Number of Independent instruction to be placed is: = (Number of Execution cycle) * 2.

It is assumed that ‘compare’ requires 3 clock cycles for execution. Hence, the number of independent instructions to be placed after the ‘compare’ instruction in the program is 3 * 2 = 6.

Program contains only 3 independent instructions. Three more independent instructions are required. Hence, for this small program fragment, 3 dummy instructions are used. It has to be noted that in the above program fragment the instructions “add H, G”, “outport G” and “outport H” are used as dummy instructions. These instructions which are independent of the compare result are included because it is assumed that the result of the compare instruction is available only after 6 clock cycles.

According to this, the predicate code can be arranged as given below.


```
        Inport A
        Inport D
        Cmpult D, A, P0, P1
        Inport B
        Inport C
        Inport E
        Add H, G
        Outport G
        Outport H
P0:     And B, E
P0:     Or C, D
P1:     Add B, E
P1:     Sub C, D
        Outport B
        Outport C
```
Instructions Independent on Compare result
Predicated Instructions
Non-Predicated Instructions

Aligned Predicated Code

Compare result, set or reset the predicate register specified in the compare, which is P0 according to the result. P1 is complement of P0.

If (D < A) then the predicate register P0 i.e., P0reg of Predicate File is set to ‘1’ and the other predicate register P1 i.e., P1reg is reset to ‘0’. If (D > A) then the P0reg is reset to ‘0’, P1reg is set to ‘1’. This is the case in a predicated architecture when ‘compare’ is executed.

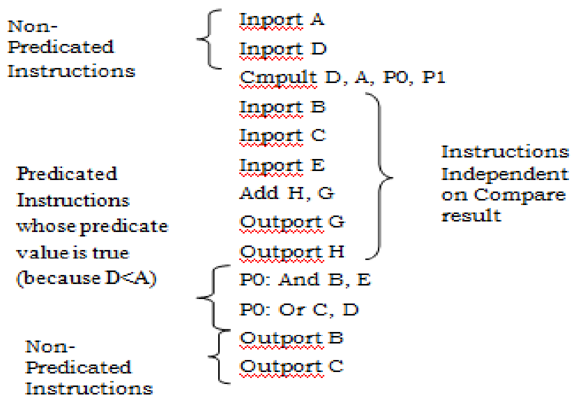
‘And’ and ‘or’ instructions are the predicated instructions, which is dependent on [P0] predicate value.

‘Add’ and ‘Sub’ instructions are also the predicated instructions dependent of [P1] predicate value.

Instruction Sequence from the predicate filter to the pipeline if (D < A):

The proposed predicate filter design allows only non-predicated instructions and predicated instruction whose predicate value are evaluated as true into the pipeline.

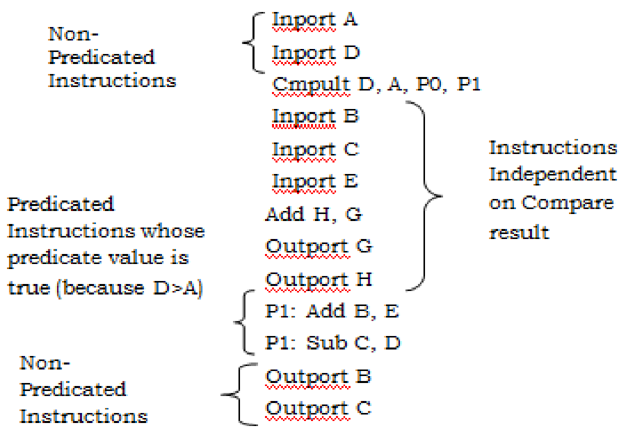
Consider that (D < A), then the program instructions that come out of the Predicate Filter are:


```
Inport A
Inport D
Cmpult D, A, P0, P1
Inport B
Inport C
Inport E
Add H, G
Outport G
Outport H
P0: And B, E
P0: Or C, D
Outport B
Outport C
```
Non-Predicated Instructions
Instructions Independent on Compare result
Predicated Instructions whose predicate value is true (because D<A)
Non-Predicated Instructions

Output of the predicate filter if (D < A)

Instruction Sequence from the predicate filter to the Pipeline if (D>A):

If (D>A), then the Predicate filter, outputs the instruction sequence as follows:


Non-Predicated Instructions
Inport A
Inport D
Cmpult D, A, P0, P1
Inport B
Inport C
Inport E
Add H, G
Outport G
Outport H
Instructions Independent on Compare result
Predicated Instructions whose predicate value is true (because D>A)
P1: Add B, E
P1: Sub C, D
Non-Predicated Instructions
Outport B
Outport C

Output of the predicate filter if (D > A)

4.1 Simulation Result

The table 1 gives the hexadecimal code for the instructions used.

Hexadecimal code (Machine Code)	Program Instructions
00001238h	Inport A
00004038h	Inport D
00211424h	Cmpult D, A, P0, P1
00002138h	Inport B
00003438h	Inport C
00005038h	Inport E
00006708h	Add H, G
00000639h	Outport G
00000739h	Outport H
8800520Ah	P0: And B, E
8800430Bh	P0: Or C, D
90005208h	P1: Add B, E
90004309h	P1: Sub C, D
00000239h	Outport B
00000339h	Outport C

Table 1: Hexadecimal codes

The simulation result for the above mentioned code is depicted. It can be observed that predicated false instructions are filtered out.

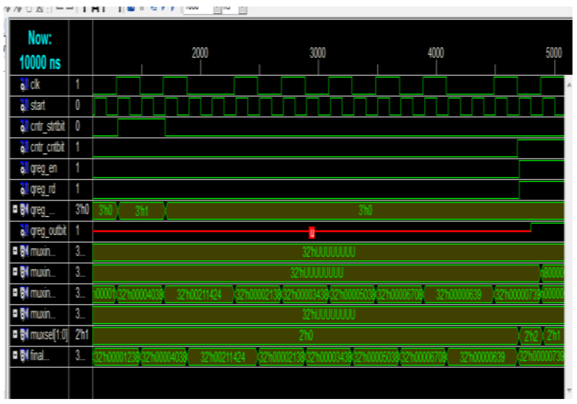


Fig 2a Simulation result if (D>A)

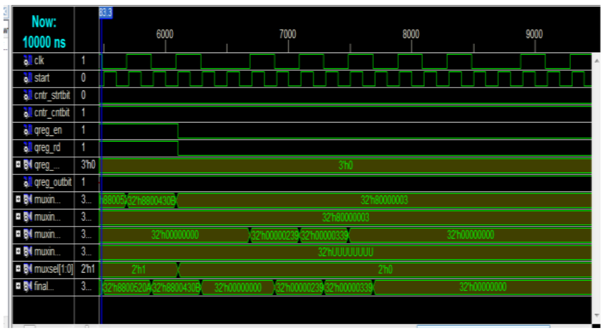


Fig 2b Continued simulation output if (D>A)

Fig 2a and Fib 2b shows the simulation result for the above program fragment if (D>A).This shows that only non-predicated instructions and predicate true instructions (And B,E and Or C,D) are coming out of the predicate filter. Only these instructions sent for the execution through pipelined queue.

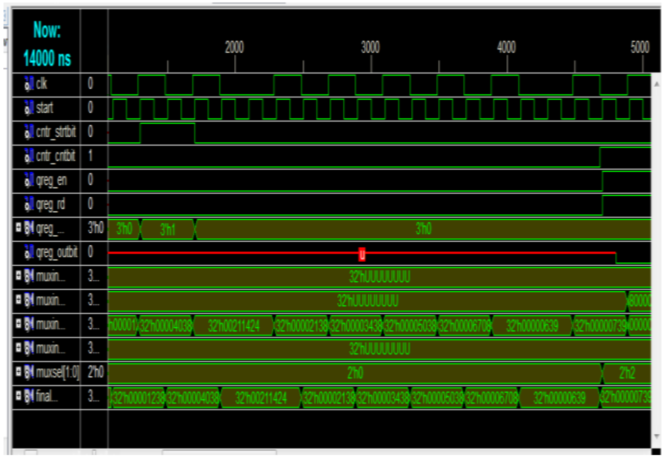


Fig 3a Simulation result if (D<A)

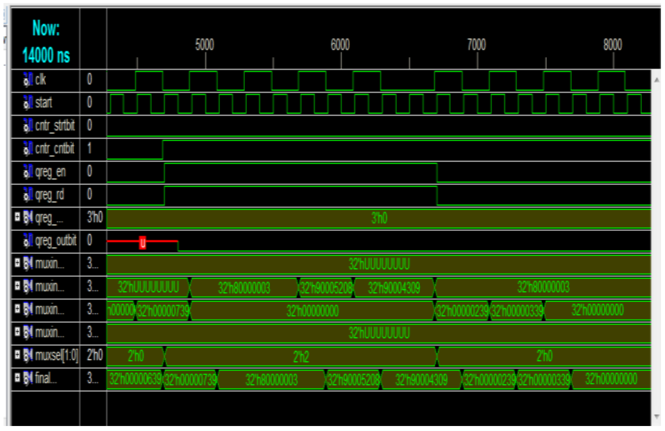


Fig 3b Continued simulation output if (D<A)

Fig 3a and 3b shows the simulation result for the above program if (D<A). This result shows that only non-predicated instructions and predicated true instructions are coming out of the predicate filter.

V. Comparative Study Of Pipelined Processor With And Without Predicate Filter

The effect of adding the predicate filter to a pipelined processor on the execution speed of the instruction is explained by taking a program fragment. The output instructions from the predicate filter are given to pipelined processor sequentially.

Example:

Consider the program which stores ‘and’ result in B, ‘or’ result in C if (A > D) and stores ‘add’ result in B, ‘sub’ result in C if (A < D).

```
 The code fragment will be
 If (A > D) then
{          B = B & E;
           C = C || D;         }
 Else
   {       B = B + E;
           C = C - D;     }
```

Instruction scheduling in a pipelined processor without predicate Filter:

For the above fragment the predicated code without predicate filter is shown in Fig 4 below. The number of clock cycles needed is also calculated.

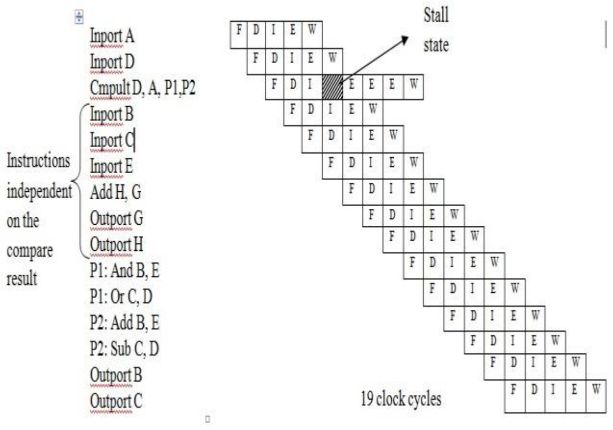


Fig 4: Pipelining without predicate filter

Program requires 6 independent instructions, to compare the result with the instruction scheduling with predicate filter. In a processor with predicate filter 6 independent instructions are needed to place them after the 'compare' instruction to avoid the waste of clock cycle during the execution of 'compare' instruction. Hence, for this small program fragment, 3 dummy instructions are used. It has to be noted that in the above program fragment the instructions "add H, G", "outport G" and "outport H" are the dummy instructions.

For the above program fragment it is assumed that all the instructions like inport, add etc take 1 clock cycle and compare instruction takes 3 clock cycles for the execution phase. In this program the compare instruction sets or resets the predicate register P1 and P2 according to the compare result. In the above program, the predicate register P1 is assumed to be true (i.e., it is assumed that D < A) and P2 is complement of P1.

The above Fig 4 shows that the complete execution of the program takes 19 clock cycles in the pipelined processor without predicated Filter.

Instruction scheduling in a pipelined processor with predicate Filter:

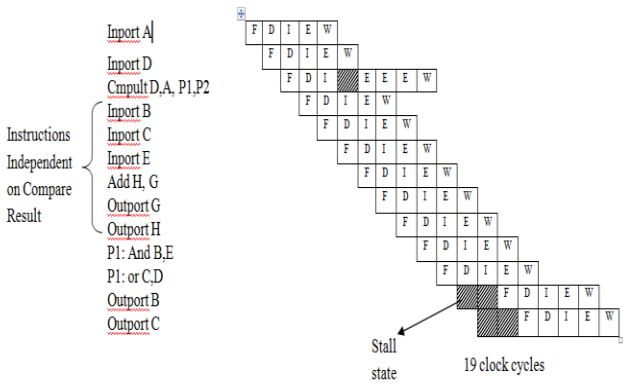


Fig 5: Pipelining with predicate filter

The instructions in the main instruction Queue are fetched sequentially. There are 2 instructions which depend on the P2 predicate register, which is assumed to be false. Two clock cycles are needed to fetch these instructions. Because of the predicated false condition these instructions are filtered out and do not enter the main pipeline stream. During this duration the pipeline stalls for 2 clock cycles which is depicted as 'up diagonal lines' in the Fig 5. This shows that there is no clock advantage with this predicate filter design.

VI. Conclusions

The design carried out shows that a predicate filter eliminates in advance the instruction that need not be executed because of invalid predicate. The output of the predicate filter containing only the instructions whose predicate value is true will be fed to the pipelined processor. This avoids instructions that should not be executed for a given predicate to enter the instruction pipeline.

An efficient utilization of the various functional units like ALU, decoder etc of the pipelined processor can be achieved due to avoidance of predicate false instruction entering the pipelined stages. It can be observed that the inclusion of the predicate filter does not require a major change in the overall design of the processor as the predicate filter is included at instruction prefetch stage.

In order to achieve speed enhancement when the predicate filter is combined with the pipelined processor further modification in the predicate filter design is needed. Various modifications to the predicate filter can be thought off which can be taken up as future work.

References

[1] Advanced **Computer Architecture**: A Design Space Approach, by · Dezso **Sima**, Terence Fountain, and Peter Kacsuk, 1997.

[2] Lam M. S. and Wilson R. P., Limits of control flow on parallelism, Proceedings of the 19th International Symposium on Computer Architecture, June 1992

[3] DeRosa J. A. and Levy H. M., An Evaluation of Branch Architectures, Proceedings of the 14th International Symposium on Computer Architecture, June 1987

[4] Riesman E. M. and Foster C. C., The Inhibition of Potential Parallelism by Conditional Jumps.